IFC Inside: Retrofitting Languages with Dynamic Information Flow Control

Extended Version

Stefan Heule[1], Deian Stefan[1], Edward Z. Yang[1], John C. Mitchell[1], and Alejandro Russo[2]**

[1] Stanford University
[2] Chalmers University

Abstract. Many important security problems in JavaScript, such as browser extension security, untrusted JavaScript libraries and safe integration of mutually distrustful websites (mash-ups), may be effectively addressed using an efficient implementation of information flow control (IFC). Unfortunately existing fine-grained approaches to JavaScript IFC require modifications to the language semantics and its engine, a non-goal for browser applications. In this work, we take the ideas of coarse-grained dynamic IFC and provide the theoretical foundation for a language-based approach that can be applied to any programming language for which external effects can be controlled. We then apply this formalism to server- and client-side JavaScript, show how it generalizes to the C programming language, and connect it to the Haskell LIO system. Our methodology offers design principles for the construction of information flow control systems when isolation can easily be achieved, as well as compositional proofs for optimized concrete implementations of these systems, by relating them to their isolated variants.

1 Introduction

Modern web content is rendered using a potentially large number of different components with differing provenance. Disparate and untrusting components may arise from browser extensions (whose JavaScript code runs alongside website code), web applications (with possibly untrusted third-party libraries), and mashups (which combine code and data from websites that may not even be aware of each other's existence.) While just-in-time combination of untrusting components offers great flexibility, it also poses complex security challenges. In particular, maintaining data privacy in the face of malicious extensions, libraries, and mashup components has been difficult.

Information flow control (IFC) is a promising technique that provides security by tracking the flow of sensitive data through a system. Untrusted code is confined so that it cannot exfiltrate data, except as per an information flow policy. Significant research has been devoted to adding various forms of IFC to

** Work partially done while at Stanford.

different kinds of programming languages and systems. In the context of the web, however, there is a strong motivation to preserve JavaScript's semantics and avoid JavaScript-engine modifications, while retrofitting it with dynamic information flow control.

The Operating Systems community has tackled this challenge (e.g., in [51]) by taking a *coarse-grained* approach to IFC: dividing an application into coarse computational units, each with a single label dictating its security policy, and only monitoring communication between them. This coarse-grained approach provides a number of advantages when compared to the fine-grained approaches typically employed by language-based systems. First, adding IFC does not require intrusive changes to an existing programming language, thereby also allowing the reuse of existing programs. Second, it has a small runtime overhead because checks need only be performed at isolation boundaries instead of (almost) every program instruction (e.g., [19]). Finally, associating a single security label with the entire computational unit simplifies understanding and reasoning about the security guarantees of the system, without reasoning about most of the technical details of the semantics of the underlying programming language.

In this paper, we present a framework which brings coarse-grained IFC ideas into a language-based setting: an information flow control system should be thought of as multiple instances of completely isolated language runtimes or *tasks*, with information flow control applied to inter-task communication. We describe a formal system in which an IFC system can be designed once and then applied to any programming language which has control over external effects (e.g., JavaScript or C with access to hardware privilege separation). We formalize this system using an approach by Matthews and Findler [28] for combining operational semantics and prove non-interference guarantees that are independent of the choice of a specific target language.

There are a number of points that distinguish this setting from previous coarse-grained IFC systems. First, even though the underlying semantic model involves communicating tasks, these tasks can be coordinated together in ways that simulate features of traditional languages. In fact, simulating features in this way is a useful *design tool* for discovering what variants of the features are permissible and which are not. Second, although completely separate tasks are semantically easy to reason about, real-world implementations often blur the lines between tasks in the name of efficiency. Characterizing what optimizations are permissible is subtle, since removing transitions from the operational semantics of a language can break non-interference. We partially address this issue by characterizing isomorphisms between the operational semantics of our abstract language and a concrete implementation, showing that if this relationship holds, then non-interference in the abstract specification carries over to the concrete implementation.

Our contributions can be summarized as follows:

– We give formal semantics for a core coarse-grained dynamic information flow control language free of non-IFC constructs. We then show how a large class

of target languages can be combined with this IFC language and prove that the result provides non-interference. (Sections 2 and 3)
- We provide a proof technique to show the non-interference of a concrete semantics for a potentially optimized IFC language by means of an isomorphism and show a class of restrictions on the IFC language that preserves non-interference. (Section 4)
- We have implemented an IFC system based on these semantics for Node.js, and we connect our formalism to another implementation based on this work for client-side JavaScript [43]. Furthermore, we outline an implementation for the C programming language and describe improvements to the Haskell LIO system that resulted from this framework. (Section 5)

2 Retrofitting Languages with IFC

Before moving on to the formal treatment of our system, we give a brief primer of information flow control and describe some example programs in our system, emphasizing the parallel between their implementation in a multi-task setting, and the traditional, "monolithic" programming language feature they simulate.

Information flow control systems operate by associating data with *labels*, and specifying whether or not data tagged with one label l_1 can flow to another label l_2 (written as $l_1 \sqsubseteq l_2$). These labels encode the desired security policy (for example, confidential information should not flow to a public channel), while the work of specifying the semantics of an information flow language involves demonstrating that impermissible flows cannot happen, a property called *non-interference* [17]. In our coarse-grained floating-label approach, labels are associated with tasks. The task label—we refer to the label of the currently executing task as the *current label*—serves to protect everything in the task's scope; all data in a task shares this common label.

As an example, here is a program which spawns a new isolated task, and then sends it a mutable reference:

$$\begin{aligned}&\textbf{let } i = {}_{\text{TI}}\lfloor \textbf{sandbox } (\textbf{blockingRecv } x, \_ \textbf{ in } {}^{\text{IT}}\lceil\, !\, {}_{\text{TI}}\lfloor x \rfloor \rceil)\rfloor\\ &\textbf{in } {}_{\text{TI}}\lfloor \textbf{send } {}^{\text{IT}}\lceil i \rceil\ l\ {}^{\text{IT}}\lceil \textbf{ref true} \rceil\rfloor\end{aligned}$$

For now, ignore the tags ${}_{\text{TI}}\lfloor \cdot \rfloor$ and ${}^{\text{IT}}\lceil \cdot \rceil$: roughly, this code creates a new **sandbox**ed task with identifier i which waits (**blockingRecv**, binding x with the received message) for a message, and then **send**s the task a mutable reference (**ref true**) which it labels l. If this operation actually shared the mutable cell between the two tasks, it could be used to violate information flow control if the tasks had differing labels. At this point, the designer of an IFC system might add label checks to mutable references, to check the labels of the reader and writer. While this solves the leak, for languages like JavaScript, where references are prevalently used, this also dooms the performance of the system.

Our design principles suggest a different resolution: when these constructs are treated as isolated tasks, each of which have their own heaps, it is obviously

the case that there is no sharing; in fact, the sandboxed task receives a dangling pointer. Even if there is only one heap, if we enforce that references not be shared, the two systems are morally equivalent. (We elaborate on this formally in Section 4.) Finally, this semantics strongly suggests that one should restrict the types of data which may be passed between tasks (for example, in JavaScript, one might only allow JSON objects to be passed between tasks, rather than general object structures).

Existing language-based, coarse-grained IFC systems [20, 41] allow a sub-computation to temporarily raise the floating-label; after the sub-computation is done, the floating-label is restored to its original label. When this occurs, the enforcement mechanism must ensure that information does not leak to the (less confidential) program continuation. The presence of exceptions adds yet more intricacies. For instance, exceptions should not automatically propagate from a sub-computation directly into the program continuation, and, if such exceptions are allowed to be inspected, the floating-label at the point of the exception-raise must be tracked alongside the exception value [18, 20, 41]. In contrast, our system provides the same flexibility and guarantees with no extra checks: tasks are used to execute sub-computations, but the mere definition of isolated tasks guarantees that (a) tasks only transfer data to the program continuation by using inter-task communication means, and (b) exceptions do cross tasks boundaries automatically.

2.1 Preliminaries

Our goal now is to describe how to take a **target language** with a formal operational semantics and combine it with an *information flow control language*. For example, taking ECMAScript as the target language and combining it with our IFC language should produce the formal semantics for the core part of COWL [43]. In this presentation, we use a simple, untyped lambda calculus with mutable references and fixpoint in place of ECMAScript to demonstrate some the key properties of the system (and, because the embedding does not care about the target language features); we discuss the proper embedding in more detail in Section 5.

Notation We have typeset nonterminals of the target language using **bold font** while the nonterminals of the IFC language have been typeset with *italic font*. Readers are encouraged to view a color copy of this paper, where target language nonterminals are colored **red** and IFC language nonterminals are colored *blue*.

2.2 Target Language: Mini-ES

In Fig. 1, we give a simple, untyped lambda calculus with mutable references and fixpoint, prepared for combination with an information flow control language. The presentation is mostly standard, and utilizes Felleisen-Hieb reduction semantics [16] to define the operational semantics of the system. One peculiarity is that our language defines an evaluation context **E**, but, the evaluation rules

$$
\begin{array}{l}
\mathbf{v} ::= \lambda \mathbf{x}.\mathbf{e} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{a} \\
\mathbf{e} ::= \mathbf{v} \mid \mathbf{x} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid \mathbf{fix}\ \mathbf{e} \\
\mathbf{E} ::= [\cdot]_{\mathbf{T}} \mid \mathbf{E}\ \mathbf{e} \mid \mathbf{v}\ \mathbf{E} \mid \mathbf{if}\ \mathbf{E}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{E} \mid !\mathbf{E} \mid \mathbf{E} := \mathbf{e} \mid \mathbf{v} := \mathbf{E} \mid \mathbf{fix}\ \mathbf{E} \\
\mathbf{e}_1; \mathbf{e}_2 \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1\ \mathbf{where}\ \mathbf{x} \notin \mathcal{FV}(\mathbf{e}_2) \\
\mathbf{let}\ \mathbf{x} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2 \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1
\end{array}
$$

$$
\text{T-APP}\ \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\left[(\lambda x.\mathbf{e})\ \mathbf{v}\right] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\left[\{\mathbf{v}\ /\ x\}\ \mathbf{e}\right]} \qquad \text{T-IFTRUE}\ \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\left[\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2\right] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\left[\mathbf{e}_1\right]}
$$

Fig. 1: λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. For space reasons we only show two representative reduction rules; full rules can be found in Appendix A.

have been expressed in terms of a different evaluation context $\mathcal{E}_{\mathbf{\Sigma}}$; Here, we follow the approach of Matthews and Findler [28] in order to simplify combining semantics of multiple languages. To derive the usual operational semantics for this language, the evaluation context merely needs to be defined as $\mathcal{E}_{\mathbf{\Sigma}}\left[\mathbf{e}\right] \triangleq \mathbf{\Sigma}, \mathbf{E}\left[\mathbf{e}\right]$. However, when we combine this language with an IFC language, we reinterpret the meaning of this evaluation context.

In general, we require that a target language be expressed in terms of some global machine state $\mathbf{\Sigma}$, some evaluation context $\mathbf{E}$, some expressions $\mathbf{e}$, some set of values $\mathbf{v}$ and a *deterministic* reduction relation on full configurations $\mathbf{\Sigma} \times \mathbf{E} \times \mathbf{e}$.

2.3 IFC Language

As mentioned previously, most modern, dynamic information flow control languages encode policy by associating a label with data. Our embedding is agnostic to the choice of labeling scheme; we only require the labels to form a lattice [12] with the partial order $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$. In this paper, we simply represent labels with the metavariable l, but do not discuss them in more detail. To enforce labels, the IFC monitor inspects the current label before performing a read or a write to decide whether the operation is permitted. A task can only write to entities that are at least as sensitive. Similarly, it can only read from entities that are less sensitive. However, as in other floating-label systems, this current label can be raised to allow the task to read from more sensitive entities at the cost of giving up the ability to write to others.

In Fig. 2, we give the syntax and *single-task* evaluation rules for a minimal information flow control language. Ordinarily, information flow control languages are defined by directly stating a base language plus information flow control operators. In contrast, our language is purposely minimal: it does not have sequencing operations, control flow, or other constructs. However, it contains support for the following core information flow control features:

- First-class labels, with label values l as well as operations for computing on labels ($\sqsubseteq$, $\sqcup$ and $\sqcap$).

– Operations for inspecting (**getLabel**) and modifying (**setLabel**) the current label of the task (a task can only increase its label).
– Operations for non-blocking inter-task communication (**send** and **recv**), which interact with the global store of per-task message queues Σ.
– A sandboxing operation used to spawn new isolated tasks. In concurrent settings **sandbox** corresponds to a fork-like primitive, whereas in a sequential setting, it more closely resembles computations which might temporarely raise the current floating-label [20, 39].

These operations are all defined with respect to an evaluation context $\mathcal{E}_{\Sigma}^{i,l}$ that represents the context of the current task. The evaluation context has three important pieces of state: the global message queues Σ, the current label l and the task ID i.

We note that first-class labels, tasks (albeit named differently), and operations for inspecting the current label are essentially universal to all floating-label systems. However, our choice of communication primitives is motivated by those present in browsers, namely `postMessage` [47]. Of course, other choices, such as blocking communication or labeled channels, are possible.

These asynchronous communication primitives are worth further discussion. When a task is sending a message using **send**, it also labels that message with a label l' (which must be at or above the task's current label l). Messages can only be received by a task if its current label is at least as high as the label of the message. Specifically, receiving a message using **recv** x_1, x_2 **in** e_1 **else** e_2 binds the message and the sender's task identifier to local variables x_1 and x_2, respectively, and then executes e_1. Otherwise, if there are no messages, that task continues its execution with e_2. We denote the filtering of the message queue by $\Theta \preceq l$, which is defined as follows. If Θ is the empty list **nil**, the function is simply the identity function, i.e., $\mathbf{nil} \preceq l = \mathbf{nil}$, and otherwise:

$$((l', i, e), \Theta) \preceq l = \begin{cases} (l', i, e), (\Theta \preceq l) & \text{if } l' \sqsubseteq l \\ \Theta \preceq l & \text{otherwise} \end{cases}$$

This ensures that tasks cannot receive messages that are more sensitive than their current label would allow.

2.4 The Embedding

Fig. 3 provides all of the rules responsible for actually carrying out the embedding of the IFC language within the target language. The most important feature of this embedding is that every task maintains its own copy of the target language global state and evaluation context, thus enforcing isolation between various tasks. In more detail:

– We extend the values, expressions and evaluation contexts of both languages to allow for terms in one language to be embedded in the other, as in [28]. In the target language, an IFC expression appears as $_{\mathrm{TI}}\lfloor e \rfloor$ ("Target-outside, IFC-inside"); in the IFC language, a target language expression appears as $^{\mathrm{IT}}\lceil \mathbf{e} \rceil$ ("IFC-outside, target-inside").

$$
\begin{array}{lll}
v & ::= i \mid l \mid \mathbf{true} \mid \mathbf{false} \mid \langle\rangle & \otimes \quad ::= \sqsubseteq \mid \sqcup \mid \sqcap \\
e & ::= v \mid x \mid e \otimes e \mid \mathbf{getLabel} \mid \mathbf{setLabel}\ e \mid \mathbf{taskId} \mid \mathbf{sandbox}\ e & \\
 & \quad \mid \mathbf{send}\ e\ e\ e \mid \mathbf{recv}\ x, x\ \mathbf{in}\ e\ \mathbf{else}\ e & \\
E & ::= [\cdot]_I \mid E \otimes e \mid v \otimes E \mid \mathbf{setLabel}\ E \mid \mathbf{send}\ E\ e\ e \mid \mathbf{send}\ v\ E\ e \mid \mathbf{send}\ v\ v\ E & \\
\theta & ::= (l, i\ e) & \Theta ::= \mathbf{nil} \mid \theta, \Theta \qquad \Sigma ::= \emptyset \mid \Sigma\,[i \mapsto \Theta]
\end{array}
$$

$$
\text{I-GETTASKID}\ \frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{taskId}] \to \mathcal{E}^{i,l}_{\Sigma}\,[i]} \qquad \text{I-GETLABEL}\ \frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{getLabel}] \to \mathcal{E}^{i,l}_{\Sigma}\,[l]} \qquad \text{I-LABELOP}\ \frac{[\![l_1 \otimes l_2]\!] = v}{\mathcal{E}^{i,l}_{\Sigma}\,[l_1 \otimes l_2] \to \mathcal{E}^{i,l}_{\Sigma}\,[v]}
$$

$$
\text{I-SEND}\ \frac{l \sqsubseteq l' \qquad \Sigma(i') = \Theta \qquad \Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{send}\ i'\ l'\ v] \to \mathcal{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}
$$

$$
\text{I-RECV}\ \frac{(\Sigma(i) \preceq l) = \theta_1, ..., \theta_k, (l', i', v) \qquad \Sigma' = \Sigma\,[i \mapsto (\theta_1, ..., \theta_k)]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \to \mathcal{E}^{i,l}_{\Sigma'}\,[\{\, v \,/\, x_1, i' \,/\, x_2 \,\}\ e_1]}
$$

$$
\text{I-NORECV}\ \frac{\Sigma(i) \preceq l = \mathbf{nil} \qquad \Sigma' = \Sigma\,[i \mapsto \mathbf{nil}]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \to \mathcal{E}^{i,l}_{\Sigma'}\,[e_2]} \qquad \text{I-SETLABEL}\ \frac{l \sqsubseteq l'}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{setLabel}\ l'] \to \mathcal{E}^{i,l'}_{\Sigma}\,[\langle\rangle]}
$$

Fig. 2: IFC language with all single-task operations.

$$
\begin{array}{llll}
v & ::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{v} \rceil & \mathbf{v} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor v \rfloor & \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}] \triangleq \Sigma; \langle \mathbf{\Sigma}, E[\mathbf{e}]_{\mathbf{T}} \rangle^{i}_{l}, \ldots \\
e & ::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{e} \rceil & \mathbf{e} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor e \rfloor & \mathcal{E}^{i,l}_{\Sigma}\,[e] \triangleq \Sigma; \langle \mathbf{\Sigma}, E[e]_{I} \rangle^{i}_{l}, \ldots \\
E & ::= \cdots \mid {}^{\mathrm{IT}}\lceil \mathbf{E} \rceil & \mathbf{E} ::= \cdots \mid {}_{\mathrm{TI}}\lfloor E \rfloor & \mathcal{E}\,[e] \to \Sigma; t, \ldots \triangleq \mathcal{E}\,[e] \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{step}}(t, \ldots)
\end{array}
$$

$$
\text{I-SANDBOX}\ \frac{\begin{array}{c}\Sigma' = \Sigma\,[i' \mapsto \mathbf{nil}] \\ \mathbf{\Sigma}' = \kappa\,(\mathbf{\Sigma}) \qquad t_1 = \langle \mathbf{\Sigma}, E[i'] \rangle^{i}_{l} \qquad t_{\mathrm{new}} = \langle \mathbf{\Sigma}', e \rangle^{i'}_{l} \qquad \mathrm{fresh}(i')\end{array}}{\Sigma; \langle \mathbf{\Sigma}, E[\mathbf{sandbox}\ e]_I \rangle^{i}_{l}, \ldots \overset{\alpha}{\hookrightarrow} \Sigma'; \alpha_{\mathrm{sandbox}}(t_1, \ldots, t_{\mathrm{new}})}
$$

$$
\text{I-DONE}\ \frac{}{\Sigma; \langle \mathbf{\Sigma}, v \rangle^{i}_{l}, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{done}}(\langle \mathbf{\Sigma}, v \rangle^{i}_{l}, \ldots)} \qquad \text{I-NOSTEP}\ \frac{\Sigma; t, \ldots \overset{\alpha}{\not\hookrightarrow}}{\Sigma; t, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\mathrm{noStep}}(t, \ldots)}
$$

$$
\text{I-BORDER}\ \frac{}{\mathcal{E}^{i,l}_{\Sigma}\left[{}^{\mathrm{IT}}\lceil {}_{\mathrm{TI}}\lfloor e \rfloor \rceil\right] \to \mathcal{E}^{i,l}_{\Sigma}\,[e]} \qquad \text{T-BORDER}\ \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\left[{}_{\mathrm{TI}}\lfloor {}^{\mathrm{IT}}\lceil \mathbf{e} \rceil \rfloor\right] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}]}
$$

Fig. 3: The embedding $L_{\mathrm{IFC}}(\alpha, \lambda)$, where $\lambda = (\mathbf{\Sigma}, \mathbf{E}, \mathbf{e}, \mathbf{v}, \to)$

$$
\begin{array}{ll}
\mathrm{RR}_{\mathrm{step}}(t_1, t_2, \ldots) & = t_2, \ldots, t_1 \\
\mathrm{RR}_{\mathrm{done}}(t_1, t_2, \ldots) & = t_2, \ldots \\
\mathrm{RR}_{\mathrm{noStep}}(t_1, t_2, \ldots) & = t_2, \ldots \\
\mathrm{RR}_{\mathrm{sandbox}}(t_1, t_2, \ldots) & = t_2, \ldots, t_1
\end{array}
\qquad
\begin{array}{ll}
\textsc{Seq}_{\mathrm{step}}(t_1, t_2, \ldots) & = t_1, t_2, \ldots \\
\textsc{Seq}_{\mathrm{noStep}}(t_1, t_2, \ldots) & = t_1, t_2, \ldots \\
\textsc{Seq}_{\mathrm{done}}(t) & = t \\
\textsc{Seq}_{\mathrm{done}}(t_1, t_2, \ldots) & = t_2, \ldots \\
\textsc{Seq}_{\mathrm{sandbox}}(t_1, t_2, \ldots, t_n) & = t_n, t_1, t_2, \ldots
\end{array}
$$

Fig. 4: Scheduling policies (concurrent round robin on the left, sequential on the right).

- We reinterpret $\mathcal{E}$ to be evaluation contexts on task lists, providing definitions for $\mathcal{E}_{\Sigma}$ and $\mathcal{E}^{i,l}_{\Sigma}$. These rules only operate on the first task in the task list, which by convention is the only task executing.
- We reinterpret $\rightarrow$, an operation on a single task, in terms of $\hookrightarrow$, operation on task lists. The correspondence is simple: a task executes a step and then is rescheduled in the task list according to schedule policy α. Fig. 4 defines two concrete schedulers.
- Finally, we define some rules for scheduling, handling sandboxing tasks (which interact with the state of the target language), and intermediating between the borders of the two languages.

The I-SANDBOX rule is used to create a new isolated task that executes separately from the existing tasks (and can be communicated with via **send** and **recv**). When the new task is created, there is the question of what the target language state of the new task should be. Our rule is stated generically in terms of a function κ. Conservatively, κ may be simply thought of as the identity function, in which case the semantics of **sandbox** are such that the state of the target language is *cloned* when sandboxing occurs. However, this is not necessary: it is also valid for κ to remove entries from the state. In Section 4, we give a more detailed discussion of the implications of the choice of κ, but all our security claims will hold regardless of the choice of κ.

The rule I-NOSTEP says something about configurations for which it is not possible to take a transition. The notation $c \overset{\alpha}{\not\rightarrow}$ in the premise is meant to be understood as follows: If the configuration c cannot take a step by any rule other than I-NOSTEP, then I-NOSTEP applies and the stuck task gets removed.

Rules I-DONE and I-NOSTEP define the behavior of the system when the current thread has reduced to a value, or gotten stuck, respectively. While these definitions simply rely on the underlying scheduling policy α to modify the task list, as we describe in Sections 3 and 6, these rules (notably, I-NOSTEP) are crucial to proving our security guarantees. For instance, it is unsafe for the whole system to get stuck if a particular task gets stuck, since a sensitive thread may then leverage this to leak information through the termination channel. Instead, as our example round-robin (RR) scheduler shows, such tasks should simply be removed from the task list. Many language runtime or Operating System schedulers implement such schedulers. Moreover, techniques such as instruction-based scheduling [10, 42] can be further applied close the gap between specified semantics and implementation.

As in [28], rules T-BORDER and I-BORDER define the syntactic boundaries between the IFC and target languages. Intuitively, the boundaries respectively correspond to an upcall into and downcall from the IFC runtime. As an example, taking λ_{ES} as the target language, we can now define a blocking receive (inefficiently) in terms of the asynchronous **recv** as series of cross-language calls:

$$\textbf{blockingRecv}\ x_1, x_2\ \textbf{in}\ e \triangleq {}^{\text{IT}}\lceil \textbf{fix}\ (\lambda k.{}_{\text{TI}}\lfloor \textbf{recv}\ x_1, x_2\ \textbf{in}\ e\ \textbf{else}\ {}^{\text{IT}}\lceil k \rceil \rfloor)\rceil$$

For any target language λ and scheduling policy α, this embedding defines an IFC language, which we will refer to as $L_{\text{IFC}}(\alpha, \lambda)$.

3 Security Guarantees

We are interested in proving non-interference about many programming languages. This requires an appropriate definition of this notion that is language agnostic, so in this section, we present a few general definitions for what an information flow control language is and what non-interference properties it may have. In particular, we show that $L_{\text{IFC}}(\alpha, \lambda)$, with an appropriate scheduler α, satisfies non-interference [17], without making any reference to properties of λ. We state the appropriate theorems here, and provide the formal proofs in Appendix D.

3.1 Erasure Function

When defining the security guarantees of an information flow control, we must characterize what the *secret inputs* of a program are. Like other work [25, 36, 39, 40], we specify and prove non-interference using *term erasure*. Intuitively, term erasure allows us to show that an attacker does not learn any sensitive information from a program if the program behaves identically (from the attackers point of view) to a program with all sensitive data "erased". To interpret a language under information flow control, we define a function ε_l that performs erasures by mapping configurations to erased configurations, usually by rewriting (parts of) configurations that are more sensitive than l to a new syntactic construct $\bullet$. We define an information flow control language as follows:

Definition 1 (Information flow control language). *An information flow control language* $\boldsymbol{L}$ *is a tuple* $(\Delta, \hookrightarrow, \varepsilon_l)$*, where* Δ *is the type of machine configurations (members of which are usually denoted by the metavariable* c*),* $\hookrightarrow$ *is a reduction relation between machine configurations and* $\varepsilon_l : \Delta \to \varepsilon(\Delta)$ *is an erasure function parametrized on labels from machine configurations to* erased *machine configurations* $\varepsilon(\Delta)$*. Sometimes, we use* V *to refer to set of terminal configurations in* Δ*, i.e., configurations where no further transitions are possible.*

Our language $L_{\text{IFC}}(\alpha, \lambda)$ fulfills this definition as $(\Delta, \overset{\alpha}{\hookrightarrow}, \varepsilon_l)$, where $\Delta = \Sigma \times \text{List}(t)$. The set of terminal conditions V is $\Sigma \times t_V$, where $t_V \subset t$ is the type for

tasks whose expressions have been reduced to values.[3] The erased configuration $\varepsilon(\Delta)$ extends Δ with configurations containing $\bullet$, and Fig. 5 gives the precise definition for our erasure function ε_l. Essentially, a task and its corresponding message queue is completely erased from the task list if its label does not flow to the attacker observation level l. Otherwise, we apply the erasure function homomorphically and remove any messages from the task's message queue that are more sensitive than l.

$$\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet) \text{ (map } \varepsilon_l \ ts)$$

$$\varepsilon_l(\langle \Sigma, e \rangle^i_{l'}) = \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \varepsilon_l(\Sigma), \varepsilon_l(e) \rangle^i_{l'} & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Theta) = \Theta \preceq l \qquad \varepsilon_l(\emptyset) = \emptyset$$

Fig. 5: Erasure function for tasks, queue maps, message queues, and configurations. In all other cases, including target-language constructs, ε_l is applied homomorphically. Note that $\varepsilon_l(e)$ is always equal to e (and similar for Σ) in this simple setting. However, when the IFC language is extended with more constructs as shown in Section 6, then this will no longer be the case.

The definition of an erasure function is quite important: it captures the attacker model, stating what can and cannot be observed by the attacker. In our case, we assume that the attacker cannot observe sensitive tasks or messages, or even the number of such entities. While such assumptions are standard [8, 40], our definitions allow for stronger attackers that may be able to inspect resource usage.[4]

3.2 Non-Interference

Given an information flow control language, we can now define non-interference. Intuitively, we want to make statements about the attacker's observational power at some security level l. This is done by defining an equivalence relation called l-equivalence on configurations: an attacker should not be able to distinguish two configurations that are l-equivalent. Since our erasure function captures what an attacker can or cannot observe, we simply define this equivalence as the syntactic-equivalence of erased configurations [40].

Definition 2 (l-equivalence). *In a language $(\Delta, \hookrightarrow, \varepsilon_l)$, two machine configurations $c, c' \in \Delta$ are considered l-equivalent, written as $c \approx_l c'$, if $\varepsilon_l(c) = \varepsilon_l(c')$.*

[3] Here, we abuse notation by describing types for configuration parts using the same metavariables as the "instance" of the type, e.g., t for the type of task.

[4] We believe that we can extend $L_{\text{IFC}}(\alpha, \lambda)$ to such models using the resource limits techniques of [48]. We leave this extension to future work.

We can now state that a language satisfies non-interference if an attacker at level l cannot distinguish the runs of any two l-equivalent configurations. This particular property is called termination sensitive non-interference (TSNI). Besides the obvious requirement to not leak secret information to public channels, this definition also requires the termination of public tasks to be independent of secret tasks. Formally, we define TSNI as follows:

Definition 3 (Termination Sensitive Non-Interference (TSNI)). *A language $(\Delta, \hookrightarrow, \varepsilon_l)$ satisfies termination sensitive non-interference if for any label l, and configurations $c_1, c_1', c_2 \in \Delta$, if*

$$c_1 \approx_l c_2 \qquad \textit{and} \qquad c_1 \hookrightarrow^* c_1' \tag{1}$$

then there exists a configuration $c_2' \in \Delta$ such that

$$c_1' \approx_l c_2' \qquad \textit{and} \qquad c_2 \hookrightarrow^* c_2' \ . \tag{2}$$

In other words, if we take two l-equivalent configurations, then for every intermediate step taken by the first configuration, there is a corresponding number of steps that the second configuration can take to result in a configuration that is l-equivalent to the first resultant configuration. By symmetry, this applies to all intermediate steps from the second configuration as well. We remark that this notion of non-interfernce is similar to *progress sensitive non-interference (PSNI)*, which accounts for leakage via progress (or termination) channels, as used for static systems [29].

Our language satisfies TSNI (and thus PSNI) under the round-robin scheduler RR of Fig. 4.

Theorem 1 (Concurrent IFC language is TSNI). *For any target language λ, $L_{IFC}(\mathrm{RR}, \lambda)$ satisfies TSNI.*

In general, however, non-interference will not hold for an arbitrary scheduler α. For example, $L_{\mathrm{IFC}}(\alpha, \lambda)$ with a scheduler that inspects a sensitive task's current state when deciding which task to schedule next will in general break non-interference [4, 35].

However, even non-adversarial schedulers are not always safe. Consider, for example, the sequential scheduling policy Seq given in Fig. 4. It is easy to show that $L_{\mathrm{IFC}}(\textsc{Seq}, \lambda)$ does not satisfy TSNI: consider a target language similar to λ_{ES} with an additional expression terminal $\Uparrow$ that denotes a divergent computation, i.e., $\Uparrow$ always reduces to $\Uparrow$ and a simple label lattice $\{\mathsf{pub}, \mathsf{sec}\}$ such that $\mathsf{pub} \sqsubseteq \mathsf{sec}$, but $\mathsf{sec} \not\sqsubseteq \mathsf{pub}$. Consider the following two configurations in this language:

$$c_1 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{if false then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_2, e \rangle^2_{\mathsf{pub}}$$
$$c_2 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{if true then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_2, e \rangle^2_{\mathsf{pub}}$$

These two configurations are pub-equivalent, but c_1 will reduce (in two steps) to $c_1' = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{true} \rceil \rangle^2_{\mathsf{pub}}$, whereas c_2 will not make any progress. Suppose that

e is a computation that writes to a pub channel,[5] then the sec task's decision to diverge or not is directly leaked to a public entity.

To accommodate for sequential languages, or cases where a weaker guarantee is sufficient, we consider an alternative non-interference property called termination insensitive non-interference (TINI). This property can also be upheld by sequential languages at the cost of leaking through (non)-termination [3].

Definition 4 (Termination insensitive non-interference (TINI)). *A language $(\Delta, V, \hookrightarrow, \varepsilon_l)$ is termination insensitive non-interfering if for any label l, and configurations $c_1, c_2 \in \Delta$ and $c_1', c_2' \in V$, it holds that*

$$(c_1 \approx_l c_2 \;\wedge\; c_1 \hookrightarrow^* c_1' \;\wedge\; c_2 \hookrightarrow^* c_2') \implies c_1' \approx_l c_2'$$

TINI states that if we take two l-equivalent configurations, and both configurations reduce to final configurations (i.e., configurations for which there are no possible further transitions), then the end configurations are also l-equivalent. We highlight that this statement is much weaker than TSNI: it only states that terminating programs do not leak sensitive data, but makes no statement about non-terminating programs.

As shown by compilers [32, 37], interpreters [19], and libraries [36, 39], TINI is useful for sequential settings. In our case, we show that our IFC language with the sequential scheduling policy Seq satisfies TINI.

Theorem 2 (Sequential IFC language is TINI). *For any target language λ, $L_{IFC}(\text{Seq}, \lambda)$ satisfies TINI.*

4 Isomorphisms and Restrictions

The operational semantics we have defined in the previous section satisfy non-interference by design. We achieve this general statement that works for a large class of languages by having different tasks executing completely isolated from each other, such that every task has its own state. In some cases, this strong separation is desirable, or even necessary. Languages like C provide direct access to memory locations without mechanisms in the language to achieve a separation of the heap. On the other hand, for other languages, this strong isolation of tasks can be undesirable, e.g., for performance reasons. For instance, for the language λ_{ES}, our presentation so far requires a separate heap per task, which is not very practical. Instead, we would like to more tightly couple the integration of the target and IFC languages by reusing existing infrastructure. In the running example, a concrete implementation might use a single global heap. More precisely, instead of using a configuration of the form $\Sigma; \langle \mathbf{\Sigma}_1, e_1 \rangle_{l_1}^{i_1}, \langle \mathbf{\Sigma}_2, e_2 \rangle_{l_2}^{i_2} \ldots$ we would like a single global heap as in $\Sigma; \mathbf{\Sigma}; \langle e_1 \rangle_{l_1}^{i_1}, \langle e_2 \rangle_{l_2}^{i_2}, \ldots$

If the operational rules are adapted naïvely to this new setting, then non-interference can be violated: as we mentioned earlier, shared mutable cells could

[5] Though we do not model labeled channels, extending the calculus with such a feature is straightforward, see Section 6.

be used to leak sensitive information. What we would like is a way of characterizing safe modifications to the semantics which preserve non-interference. The intention of our single heap implementation is to permit efficient execution while *conceptually maintaining isolation between tasks* (by not allowing sharing of references between them). This intuition of having a different (potentially more efficient) concrete semantics that behaves like the abstract semantics can be formalized by the following definition:

Definition 5 (Isomorphism of information flow control languages). *A language $(\Delta, \hookrightarrow, \varepsilon_l)$ is isomorphic to a language $(\Delta', \hookrightarrow', \varepsilon_l')$ if there exist total functions $f : \Delta \to \Delta'$ and $f^{-1} : \Delta' \to \Delta$ such that $f \circ f^{-1} = id_\Delta$ and $f^{-1} \circ f = id_{\Delta'}$. Furthermore, f and f^{-1} are functorial (e.g., if x' R' y' then $f(x')$ R $f(y')$) over both l-equivalences and $\hookrightarrow$.*

If we weaken this restriction such that f^{-1} does not have to be functorial over $\hookrightarrow$, we call the language $(\Delta, \hookrightarrow, \varepsilon_l)$ weakly isomorphic to $(\Delta', \hookrightarrow', \varepsilon_l')$.

Providing an isomorphism between the two languages allows us to preserve (termination sensitive or insensitive) non-interference as the following two theorems state.

Theorem 3 (Isomorphism preserves TSNI). *If L is isomorphic to L' and L' satisfies TSNI, then L satisfies TSNI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TSNI, and then transporting the resulting configuration, l-equivalence and multi-step derivation back. □

Only weak isomorphism is necessary for TINI. Intuitively, this is because it is not necessary to back-translate reduction sequences in L' to L; by the definition of TINI, we have both reduction sequences in L by assumption.

Theorem 4 (Weak isomorphism preserves TINI). *If a language L is weakly isomorphic to a language L', and L' satisfies TINI, then L satisfies TINI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TINI and transporting the resulting equivalence back using functoriality of f^{-1} over l-equivalences. □

Unfortunately, an isomorphism is often too strong of a requirement. To obtain an isomorphism with our single heap semantics, we need to mimic the behavior of several heaps with a single actual heap. The interesting cases are when we sandbox an expression and when messages are sent and received. The rule for sandboxing is parametrized by the strategy κ (see Section 2), which defines what heap the new task should execute with. We have considered two choices:

– When we sandbox into an empty heap, existing addresses in the sandboxed expression are no longer valid and the task will get stuck (and then removed by I-noStep). Thus, we must rewrite the sandboxed expression so that all addresses point to fresh addresses guaranteed to not occur in the heap. Similarly, sending a memory address should be rewritten.

– When we clone the heap, we have to copy everything reachable from the sandboxed expression and replace all addresses correspondingly. Even worse, the behavior of sending a memory address now depends on whether that address existed at the time the receiving task was sandboxed; if it did, then the address should be rewritten to the existing one.

Isomorphism demands we implement this convoluted behavior, despite our initial motivation of a more efficient implementation.

4.1 Restricting the IFC Language

A better solution is to forbid sandboxed expressions as well as messages sent to other tasks to contain memory addresses in the first place. In a statically typed language, the type system could prevent this from happening. In dynamically typed languages such as λ_{ES}, we might restrict the transition for **sandbox** and **send** to only allow expressions without memory addresses.

While this sounds plausible, it is worth noting that we are modifying the IFC language semantics, which raises the question of whether non-interference is preserved. This question can be subtle: it is easy to remove a transition from a language and invalidate TSNI. Intuitively if the restriction depends on secret data, then a public thread can observe if some other task terminates or not, and from that obtain information about the secret data that was used to restrict the transition. With this in mind, we require semantic rules to get restricted only based on information observable by the task triggering them. This ensures that non-interference is preserved, as the restriction does not depend on confidential information. Below, we give the formal definition of this condition for the abstract IFC language $L_{\text{IFC}}(\alpha, \lambda)$.

Definition 6 (Restricted IFC language). *For a family of predicates $\mathcal{P}$ (one for every reduction rule), we call $L^{\mathcal{P}}_{IFC}(\alpha, \lambda)$ a restricted IFC language if its definition is equivalent to the abstract language $L_{IFC}(\alpha, \lambda)$, with the following exception: the reduction rules are restricted by adding a predicate $P \in \mathcal{P}$ to the premise of all rules other than* I-noStep*. Furthermore, the predicate P can depend only on the erased configuration $\varepsilon_l(c)$, where l is the label of the first task in the task list and c is the full configuration.*

By the following theorem, the restricted IFC language with an appropriate scheduling policy is non-interfering.

Theorem 5. *For any target language λ and family of predicates $\mathcal{P}$, the restricted IFC language $L^{\mathcal{P}}_{IFC}(\text{RR}, \lambda)$ is TSNI. Furthermore, the IFC language $L^{\mathcal{P}}_{IFC}(\text{Seq}, \lambda)$ is TINI.*

In Appendix B we give an example how this formalism can be used to show non-intereference of an implementation of IFC with a single heap.

5 Real World Languages

Our approach can be used to retrofit any language for which we can achieve isolation with information flow control. Unfortunately, controlling the external effects of a real-world language, as to achieve isolation, is language-specific and varies from one language to another.[6] Indeed, even for a single language (e.g., JavaScript), how one achieves isolation may vary according to the language runtime or embedding (e.g., server and browser).

In this section, we describe several implementations and their approaches to isolation. In particular, we describe two JavaScript IFC implementations building on the theoretical foundations of this work. Then, we consider how our formalism could be applied to the C programming language and connect it to a previous IFC system for Haskell.

5.1 JavaScript

JavaScript, as specified by ECMAScript [14], does not have any built-in functionality for I/O. For this language, which we denote by λ_{JS}, the IFC system $L_{\text{IFC}}(\text{RR}, \lambda_{\text{JS}})$ can be implemented by exposing IFC primitives to JavaScript as part of the runtime, and running multiple instances of the JavaScript virtual machine in separate OS-level threads. Unfortunately, this becomes very costly when a system, such as a server-side web application, relies on many tasks.

Luckily, this issue is not unique to our work—browser layout engines also rely on isolating code executing in separate iframes (e.g., according to the same-origin policy). Since creating an OS thread for each iframe is expensive, both the V8 and SpiderMonkey JavaScript engines provide means for running JavaScript code in isolation within a single OS thread, on disjoint sub-heaps. In V8, this unit of isolation is called a *context*; in SpiderMonkey, it is called a *compartment*. (We will use these terms interchangeably.) Each context is associated with a global object, which, by default, implements the JavaScript standard library (e.g., `Object`, `Array`, etc.). Naturally, we adopt contexts to implement our notion of tasks.

When JavaScript is embedded in browser layout engines, or in server-side platforms such as Node.js, additional APIs such as the Document Object Model (DOM) or the file system get exposed as part of the runtime system. These features are exposed by extending the global object, just like the standard library. For this reason, it is easy to modify these systems to forbid external effects when implementing an IFC system, ensuring that important effects can be reintroduced in a safe manner.

Server-side IFC for Node.js: We have implemented $L_{\text{IFC}}(\text{SEQ}, \lambda_{\text{JS}})$ for Node.js in the form of a library, without modifying Node.js or the V8 JavaScript engine. Our implementation[7] provides a library for creating new tasks, i.e., contexts

[6] Though we apply our framework to several real-world languages, it is conceivable that there are languages for which isolation cannot be easily achieved.

[7] Available at `http://github.com/deian/espectro`.

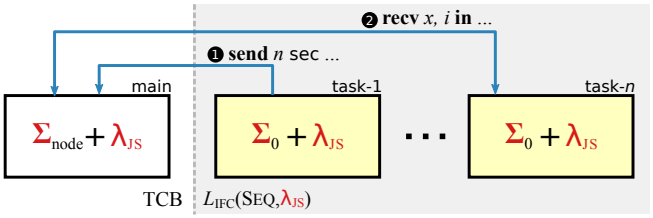

Fig. 6: This example shows how our trusted monitor (left) is used to mediate communication between two tasks for which IFC is enforced (right).

whose global object only contains the standard JavaScript library and our IFC primitives (e.g., **send** and **sandbox**). When mapped to our formal treatment, **sandbox** is defined with $\kappa(\mathbf{\Sigma}) = \mathbf{\Sigma}_0$, where $\mathbf{\Sigma}_0$ is the global object corresponding to the standard JavaScript library and our IFC primitives. These IFC operations are mediated by the trusted library code (executing as the main Node.js context), which tracks the state (current label, messages, etc.) of each task. An example for **send**/**recv** is shown in Fig. 6. Our system conservatively restricts the kinds of messages that can be exchanged, via **send** (and **sandbox**), to string values. In our formalization, this amounts to restricting the IFC language rule for **send** in the following way:

$$\text{JS-SEND}\quad \frac{\begin{array}{ll} l \sqsubseteq l' & \Sigma\,(i') = \Theta \qquad \Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta] \\ e = {}^{\text{IT}}\lceil \mathbf{e} \rceil & \mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{typeOf(}\mathbf{e}\texttt{)} \texttt{ === "string"}] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{true}] \end{array}}{\Sigma; \langle \mathbf{\Sigma}, E[\textbf{send}\ i'\ l'\ v]_I \rangle^i_l, \ldots \hookrightarrow \Sigma'; \alpha_{\text{step}}(\langle \mathbf{\Sigma}, E[\langle\rangle]_I \rangle^i_l, \ldots)}$$

Of course, we provide a convenience library which marshals JSON objects to/from strings. We remark that this is not unlike existing message-passing JavaScript APIs, e.g., `postMessage`, which impose similar restrictions as to avoid sharing references between concurrent code.

While the described system implements $L_{\text{IFC}}(\text{SEQ}, \lambda_{\text{JS}})$, applications typically require access to libraries (e.g., the file system library `fs`) that have external effects. Exposing the Node.js APIs directly to sandboxed tasks is unsafe. Instead, we implement libraries (like a labeled version of `fs`) as message exchanges between the sandboxed tasks (e.g., `task-1` in Fig. 6) and the main Node.js task that implements the IFC monitor. While this is safer than simply wrapping unsafe objects, which can potentially be exploited to access objects outside the context (e.g., as seen with ADSafe, FBJS, and Caja [26, 27, 44]), adding features such as the `fs` requires the code in the main task to ensures that labels are properly propagated and enforced. Unfortunately, while imposing such a proof burden is undesirable, this also has to be expected: different language environments expose different libraries for handling external I/O, and the correct treatment of external effects is application specific. We do not extend our formalism to account for the particular interface to the file system, HTTP client, etc., as this is specific to the Node.js implementation and does not generalize to other systems.

Client-side IFC: This work provides the formal basis for the core part of the COWL client-side JavaScript IFC system [43]. Like our Node.js implementation, COWL takes a coarse-grained approach to providing IFC for JavaScript programs. However, COWL's IFC monitor is implemented in the browser layout engine instead (though still leaving the JavaScript engine unmodified).

Furthermore, COWL repurposes existing contexts (e.g., iframes and pages) as IFC tasks, only imposing additional constraints on how they communicate. As with Node.js, at its core, the global object of a COWL task should only contain the standard JavaScript libraries and `postMessage`, whose semantics are modeled by our JS-SEND rule. However, existing contexts have objects such as the DOM, which require COWL to restrict a task's external effects. To this end, COWL mediates any communication (even via the DOM) at the context boundary.

Simply disallowing all the external effects is overly-restricting for real-world applications (e.g., pages typically load images, perform network requests, etc.). In this light, COWL allows safe network communication by associating an implicit label with remote hosts (a host's label corresponds to its origin). In turn, when a task performs a request, COWL's IFC monitor ensures that the task label can flow to the remote origin label. While the external effects of COWL can be formally modeled, we do not model them in our formalism, since, like for the Node.js case, they are specific to this system.

5.2 Haskell

Our work borrows ideas from the LIO Haskell coarse-grained IFC system [39, 40]. LIO relies on Haskell's type system and monadic encoding of effects to achieve isolation and define the IFC sub-language. Specifically, LIO provides the `LIO` monad as a way of restricting (almost all) side-effects. In the context of our framework, LIO can be understood as follows: the *pure subset* of Haskell is the target language, while the monadic subset of Haskell, operating in the `LIO` monad, is the IFC language.

Unlike our proposal, LIO originally associated labels with exceptions, in a similar style to fine-grained systems [20, 41]. In addition to being overly complex, the interaction of exceptions with clearance (which sets an upper bound on the floating label, see Appendix C.3) was incorrect: the clearance was restored to the clearance at point of the catch. Furthermore, pure exceptions (e.g., divide by zero) always percolated to trusted code, effectively allowing for denial of service attacks. The insights gained when viewing coarse-grained IFC as presented in this paper led to a much cleaner, simpler treatment of exceptions, which has now been adopted by LIO.

5.3 C

C programs are able to execute arbitrary (machine) code, access arbitrary memory, and perform arbitrary system calls. Thus, the confinement of C programs must be imposed by the underlying OS and hardware. For instance, our notion

of isolation can be achieved using Dune's hardware protection mechanisms [5], similar to Wedge [5, 7], but using an information flow control policy. Using page tables, a (trusted) IFC runtime could ensure that each task, implemented as a lightweight process, can only access the memory it allocates—tasks do not have access to any shared memory. In addition, ring protection could be used to intercept system calls performed by a task and only permit those corresponding to our IFC language (such as **getLabel** or **send**). Dune's hardware protection mechanism would allow us to provide a concrete implementation that is efficient and relatively simple to reason about, but other sandboxing mechanisms could be used in place of Dune.

In this setting, the combined language of Section 2 can be interpreted in the following way: calling from the target language to the IFC language corresponds to invoking a system call. Creating a new task with the **sandbox** system call corresponds to *forking* a process. Using page tables, we can ensure that there will be no shared memory (effectively defining $\kappa(\mathbf{\Sigma}) = \mathbf{\Sigma}_0$, where $\mathbf{\Sigma}_0$ is the set of pages necessary to bootstrap a lightweight process). Similarly, control over page tables and protection bits allows us to define a **send** system call that copies pages to our (trusted) runtime queue; and, correspondingly, a **recv** that copies the pages from the runtime queue to the (untrusted) receiver. Since C is not memory safe, conditions on these system calls are meaningless. We leave the implementation of this IFC system for C as future work.

6 Extensions and Limitations

While the IFC language presented thus far provides the basic information flow primitives, actual IFC implementations may wish to extend the minimal system with more specialized constructs. For example, COWL provides a labeled version of the XMLHttpRequest (XHR) object, which is used to make network requests. Our system can be extended with constructs such as labeled values, labeled mutable references, clearance, and privileges. For space reasons, we provide details of this, including the soundness proof with the extensions, in Appendix C. Here, we instead discuss a limitation of our formalism: the lack of external effects.

Specifically, our embedding assumes that the target language does not have any primitives that can induce external effects. As discussed in Section 5, imposing this restriction can be challenging. Yet, external effects are crucial when implementing more complex real-world applications. For example, code in an IFC browser must load resources or perform XHR to be useful.

Like labeled references, features with external effects must be modeled in the IFC language; we must reason about the precise security implications of features that otherwise inherently leak data. Previous approaches have modeled external effects by internalizing the effects as operations on labeled channels/references [40]. Alternatively, it is possible to model such effects as messages to/from certain labeled tasks, an approach taken by our Node.js implementation. These "special" tasks are trusted with access to the unlabeled primitives that can be used to perform the external effects; since the interface to these

tasks is already part of the IFC language, the proof only requires showing that this task does not leak information. Instead of restricting or wrapping unsafe primitives, COWL allow for controlled network communication at the context boundary. (By restricting the default XHR object, for example, COWL allows code to communicate with hosts according to the task's current label.)

7 Related Work

Our information flow control system is closely related to the coarse-grained information systems used in operating systems such as Asbestos [15], HiStar [51], and Flume [24], as well as language-based *floating-label IFC systems* such as LIO [39], and Breeze [20], where there is a monotonically increased label associated with threads of execution. Our treatment of termination-sensitive and termination-insensitive interference originates from Smith and Volpano [38, 46].

One information flow control technique designed to handle legacy code is secure multi-execution (SME) [13, 34]. SME runs multiple copies of the program, one per security level, where the semantics of I/O interactions is altered. Bielova et al. [6] use a transition system to describe SME, where the details of the underlying language are hidden. Zanarini et al. [50] propose a novel semantics for programs based on interaction trees [21], which treats programs as black-boxes about which nothing is known, except what can be inferred from their interaction with the environment. Similar to SME, our approach mediates I/O operations; however, our approach only runs the program once.

One of the primary motivations behind this paper is the application of information flow control to JavaScript. Previous systems retrofitted JavaScript with fine-grained IFC [18, 19, 23]. While fine-grained IFC can result in fewer false alarms and target legacy code, it comes at the cost of complexity: the system must accommodate the entirety of JavaScript's semantics [19]. By contrast, coarse-grained approaches to security tend to have simpler implications [11, 49].

The constructs in our IFC language, as well as the behavior of inter-task communication, are reminiscent of distributed systems like Erlang [2]. In distributed systems, isolation is required due to physical constraints; in information flow control, isolation is required to enforce non-interference. Papagiannis et al. [33] built an information flow control system on top of Erlang that shares some similarities to ours. However, they do not take a floating-label approach (processes can find out when sending a message failed due to a forbidden information flow), nor do they provide security proofs.

There is limited work on general techniques for retrofitting arbitrary languages with information flow control. However, one time-honored technique is to define a fundamental calculus for which other languages can be desugared into. Abadi et al. [1] motivate their core calculus of dependency by showing how various previous systems can be encoded in it. Tse and Zdancewic [45], in turn, show how this calculus can be encoded in System F via parametricity. Broberg and Sands [9] encode several IFC systems into Paralocks. However, this line of work is primarily focused on static enforcements.

8 Conclusion

In this paper, we argued that when designing a coarse-grained IFC system, it is better to start with a fully isolated, multi-task system and work one's way back to the model of a single language equipped with IFC. We showed how systems designed this way can be proved non-interferent without needing to rely on details of the target language, and we provided conditions on how to securely refine our formal semantics to consider optimizations required in practice. We connected our semantics to two IFC implementations for JavaScript based on this formalism, explained how our methodology improved an exiting IFC system for Haskell, and proposed an IFC system for C using hardware isolation. By systematically applying ideas from IFC in operating systems to programming languages for which isolation can be achieved, we hope to have elucidated some of the core design principles of coarse-grained, dynamic IFC systems.

Acknowledgements We thank the POST 2015 anonymous reviewers, Adriaan Larmuseau, Sergio Maffeis, and David Mazières for useful comments and suggestions. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by the NSF, by the AFOSR, by multiple gifts from Google, by a gift from Mozilla, and by the Swedish research agencies VR and the Barbro Oshers Pro Suecia Foundation. Deian Stefan and Edward Z. Yang were supported by the DoD through the NDSEG.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *POPL*, 1999.
[2] J. Armstrong. Making reliable distributed systems in the presence of software errors. 2003.
[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. ESORICS, 2008.
[4] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *ESORICS*, 2007.
[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, 2012.
[6] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for a browser model. In *NSS*, 2011.
[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.
[8] Boudol and Castellani. Noninterference for concurrent programs. In *ICALP*, 2001.
[9] N. Broberg and D. Sands. Paralocks: Role-based information flow control and beyond. In *POPL*, 2010.
[10] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems. In *TGC*, 2013.

[11] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, 2012.
[12] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.
[13] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *SP*, 2010.
[14] Ecma International. ECMAScript language specification. http://www.ecma.org/, 2014.
[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, 2005.
[16] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2), 1992.
[17] J. Goguen and J. Meseguer. Security policies and security Models. In *SP*, 1982.
[18] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*, 2012.
[19] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.
[20] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *SP*, 2013.
[21] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS*, 62, 1997.
[22] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*, 1996.
[23] C. Kerschbaumer, E. Hennigan, S. Brunthaler, P. Larsen, and M. Franz. Integrity considerations for secure computer systems. Technical Report 12-01, Univ. of California Irvine, 2012.
[24] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.
[25] P. Li and S. Zdancewic. Arrows for secure information flow. *TCS*, 411(19), 2010.
[26] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *CSF*, 2009.
[27] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *SP*, 2010.
[28] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, 2007.
[29] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *CCS*, 2012.
[30] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.
[31] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Comput. Syst.*, 9(4):410–442, 2000.
[32] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, 2001.

[33] I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Sh, J. Bacon, and P. Pietzuch. Enforcing user privacy in web applications using Erlang. In *W2SP*, 2010.

[34] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. In *CSF*, 2013.

[35] A. Russo and A. Sabelfeld. Securing Interaction between threads and the scheduler. In *CSFW*, 2006.

[36] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell*, 2008.

[37] V. Simonet. The Flow Caml system. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, 2003.

[38] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, 1998.

[39] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell*, 2011.

[40] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, 2012.

[41] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[42] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *ESORICS*, 2013.

[43] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting users by confining JavaScript with COWL. In *OSDI*, 2014.

[44] A. Taly, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *SP*, 2011.

[45] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *ICFP*, 2004.

[46] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, 1997.

[47] W3C. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, 2012.

[48] E. Z. Yang and D. Mazières. Dynamic space limits for Haskell. In *PLDI*, 2014.

[49] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*, 2009.

[50] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *CSF*, 2013.

[51] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

A Full Semantics for λ_{ES}

In Fig. 7 we give the full semantics for λ_{ES}. A subset of them has been given in Fig. 1 earlier in the paper.

$$
\begin{array}{l}
\mathbf{v} ::= \lambda \mathbf{x}.\mathbf{e} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{a} \\
\mathbf{e} ::= \mathbf{v} \mid \mathbf{x} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid \mathbf{fix}\ \mathbf{e} \\
\mathbf{E} ::= [\cdot]_{\mathbf{T}} \mid \mathbf{E}\ \mathbf{e} \mid \mathbf{v}\ \mathbf{E} \mid \mathbf{if}\ \mathbf{E}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{ref}\ \mathbf{E} \mid !\mathbf{E} \mid \mathbf{E} := \mathbf{e} \mid \mathbf{v} := \mathbf{E} \mid \mathbf{fix}\ \mathbf{E} \\
\mathbf{e}_1; \mathbf{e}_2 \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1\ \mathbf{where}\ \mathbf{x} \notin \mathcal{FV}(\mathbf{e}_2) \\
\mathbf{let}\ \mathbf{x} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2 \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1
\end{array}
$$

$$\text{T-APP}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[(\lambda x.\mathbf{e})\ \mathbf{v}] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\{\mathbf{v}\ /\ x\}\ \mathbf{e}]}$$

$$\text{T-IFTRUE}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}_1]}$$

$$\text{T-IFFALSE}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\ \mathbf{if}\ \mathbf{false}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}_2]}$$

$$\text{T-REF}\quad \frac{\mathrm{fresh}(\mathbf{a})}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{ref}\ \mathbf{v}] \to \mathcal{E}_{\mathbf{\Sigma}[\mathbf{a}\mapsto\mathbf{v}]}\,[\mathbf{a}]}$$

$$\text{T-DEREF}\quad \frac{(\mathbf{a}, \mathbf{v}) \in \mathbf{\Sigma}}{\mathcal{E}_{\mathbf{\Sigma}}\,[!\mathbf{a}] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{v}]}$$

$$\text{T-ASS}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{a} := \mathbf{v}] \to \mathcal{E}_{\mathbf{\Sigma}[\mathbf{a}\mapsto\mathbf{v}]}\,[\mathbf{v}]}$$

$$\text{T-FIX}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{fix}\ (\lambda x.e)] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\{\mathbf{fix}\ (\lambda x.e)\ /\ x\}\ e]}$$

Fig. 7: λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. $\mathcal{FV}$ (**e**) returns the set of free variables in expression **e**.

B Example IFC Language with a Single Heap

As a concrete instantiation of this proof technique, we show how to make implement our IFC language using a single heap and ensure its non-interference using the techniques presented. First, we can construct the restricted language $L_{\mathrm{IFC}}^{\mathcal{P}_{\mathrm{norefs}}}(\alpha, \lambda_{ES})$, where $\mathcal{P}_{\mathrm{norefs}}$ is the family of always valid predicates, except for the ones for I-SANDBOX and I-SEND, which we define as $P(e) = (\mathcal{AV}(e) = \emptyset)$ where $\mathcal{AV}(e)$ denotes the set of address variables in e. That is, we do not restrict any rules except for I-SANDBOX and I-SEND. Since P only depends on e, which is part of the current task and thus never erased w.r.t. the label of the first task, this language satisfies non-interference by Theorem 5.

The essential parts of the semantics for the concrete language with a single heap, which we call $L_{\mathrm{IFC}}^{\mathrm{Heap}}(\alpha)$, are given in Fig. 8. Most rules are straight-forward

$$\textsc{C-sandbox}\quad \frac{\mathcal{AV}(e) = \emptyset \qquad \Sigma' = \Sigma\left[i' \mapsto \mathbf{nil}\right] \qquad t_1 = \langle E[i'] \rangle_l^i \qquad t_{\text{new}} = \langle {}_{\text{TI}} \lfloor e \rfloor \rangle_l^{i'} \qquad \text{fresh}(i')}{\Sigma; \mathbf{\Sigma}; \langle E[\mathbf{sandbox}\ e]_I \rangle_{l_1}^{i_1}, \ldots \hookrightarrow \Sigma'; \mathbf{\Sigma}; \alpha_{\text{sandbox}}(t_1, \ldots, t_{\text{new}})}$$

$$\textsc{C-send}\quad \frac{\mathcal{AV}(e) = \emptyset \qquad l \sqsubseteq l' \qquad \Sigma\ (i') = \Theta \qquad \Sigma' = \Sigma\left[i' \mapsto (l', i, v), \Theta\right]}{\Sigma; \mathbf{\Sigma}; \langle E[\mathbf{send}\ i'\ l'\ v]_I \rangle_l^i, \ldots \to \Sigma; \mathbf{\Sigma}; \alpha_{\text{step}}(\langle\langle\rangle\rangle_l^i, \ldots)}$$

Fig. 8: A selection of the reduction rules for $L_{\text{IFC}}^{\text{Heap}}(\alpha)$.

translations of the rules in Figs. 2 and 3 but for a single heap. For conciseness, we only show the interesting ones. Now, we can show an isomorphism between this language and $L_{\text{IFC}}^{\mathcal{P}_{\text{norefs}}}(\alpha, \lambda_{\text{ES}})$, which (by Theorem 3 and 4) guarantees non-interference for an appropriate scheduling policy α.

To this end, we represent addresses in the concrete language as pairs $(i, \mathbf{a})$ where i is a task identifier, and $\mathbf{a}$ an address in the abstract system[8]. We also formulate the following well-formedness condition for configurations:

$$\text{wf}(c) = \forall \langle e \rangle_l^i \in c.\ \{(i', e') \in \mathcal{AV}(e) \mid i \neq i'\} = \emptyset$$

Essentially, every address in a given task must have the correct identifier as the first part of the address. It is easy to see that the initial configuration satisfies this condition, and any step in the concrete semantics preserves the condition. Therefore, we only need to consider well-formed configurations, which allows us to give the two required functions f and f^{-1} for the isomorphism. For conciseness, we only give the interesting parts of their definition, and leave out the straightforward proof that they actually provide an isomorphism.

- Addresses can be directly translated with $f((i, \mathbf{a})) = \mathbf{a}$, and $f^{-1}(\mathbf{a}) = (i, \mathbf{a})$ for an address $\mathbf{a}$ that occurs in task i.
- f splits the single heap into multiple heaps based on the i of the addresses. f^{-1} produces a single heap by translating the addresses and collapsing everything to a single store.

C Extending the Core Calculus

As mentioned in the main body of this paper, actual IFC implementations may wish to extend the minimal system with more specialized constructs. In this section we show how to extend the language with several such constructs.

[8] Note that this does not make the isomorphism trivial, as in the single heap, there is nothing preventing task 1 to access an address (2,$\mathbf{a}$). Furthermore, it is common to represent addresses in this way for efficient garbage collection of dead tasks.

C.1 Labeled values

In traditional language-based dynamic IFC systems, a label is associated with values. Hence, a program that, for example, simply writes labeled messages to a labeled log can operate on both public and sensitive values. Similarly, a task that receives a sensitive value and forwards it to another task does not have be be at a sensitive level, if the value is not inspected. In its simplest form, our coarse grained system requires that the current label of a task be at least at the level of the sensitive data to reflect the fact that such data is in scope.

If such fine-grained labeling of values is required, our base IFC system can be extended with explicitly labeled values, much like those of LIO and Breeze [20, 39]: $v ::= \cdots \mid \textbf{Labeled}\ l\ e$. Following LIO, we say that the expression e is protected by label l, while the label l itself is protected by the task's current label. The label of such values can be inspected the task without requiring the current label to be raised. However, when a task wishes to inspect the protected value e, it must first raise its label to at least l to reflect that it is incorporating data at such sensitivity level in its scope. When creating labeled values the label l must be above the current label; otherwise it cannot be said that protection has been transferred from the current label to l.

In Fig. 9, we formally show how to add this extension to the language. We assume that the constructor **Labeled** is not part of the surface syntax, but rather an internal construct.

$$
\begin{array}{l}
v ::= \cdots \mid \textbf{Labeled}\ l\ e \\
e ::= \cdots \mid \textbf{label}\ e\ e \mid \textbf{unlabel}\ e \mid \textbf{labelOf}\ e \\
E ::= \cdots \mid \textbf{label}\ E\ e \mid \textbf{unlabel}\ E \mid \textbf{labelOf}\ E
\end{array}
$$

$$
\textsc{I-label}\ \frac{l \sqsubseteq l'}{\mathcal{E}_{\Sigma}^{i,l}\ [\textbf{label}\ l'\ e] \rightarrow \mathcal{E}_{\Sigma}^{i,l}\ [\textbf{Labeled}\ l'\ e]}
\qquad
\textsc{I-unlabel}\ \mathcal{E}_{\Sigma}^{i,l}\ [\textbf{unlabel}\ (\textbf{Labeled}\ l'\ e)] \rightarrow \mathcal{E}_{\Sigma}^{i,l \sqcup l'}\ [e]
$$

$$
\textsc{I-labelOf}\ \frac{}{\mathcal{E}_{\Sigma}^{i,l}\ [\textbf{labelOf}\ (\textbf{Labeled}\ l'\ e)] \rightarrow \mathcal{E}_{\Sigma}^{i,l}\ [l']}
$$

Fig. 9: Syntax and semantics for labeled values. These rules are understood to be an addition to the existing rules given earlier.

C.2 Labeled mutable references/variables/channels

Extending the calculus with other labeled features, such as references, mutable variables (MVars) [22], or channels, can be done in a similar manner: these references are implemented in the IFC language, separately from any preexisting

notions of mutable references in the target language. There is some minor additional state to track: specifically, by amending Σ, as in [39, 40], we can allow threads to use these constructs to synchronize, or communicate with constructs other than **send**/**recv** in a safe manner. For example, when extending the calculus with labeled references, Σ additionally contains a store that maps addresses to a value and a label which can be read and written to by different tasks through a labeled reference implementations.

In Fig. 10 details labeled references formally. The construct a_l is internal in the labeled reference implementation, and not part of the surface syntax. The changes to the language for labeled values and references require us to update the erasure function ε_l, whose full definition is shown in Fig. 11.

$$
\begin{array}{rcl}
v & ::= & \cdots \mid a_l \\
e & ::= & \cdots \mid \textbf{new}\ e\ e \mid \textbf{read}\ e \mid \textbf{write}\ e\ e \\
E & ::= & \cdots \mid \textbf{new}\ E\ e \mid \textbf{new}\ l\ E \mid \textbf{read}\ E \\
 & & \mid \textbf{write}\ E\ e \mid \textbf{write}\ a_l\ E \\
\Sigma & ::= & \cdots \mid \Sigma\,[a_l \mapsto v]
\end{array}
$$

$$
\text{I-NEW}\quad \frac{l \sqsubseteq l' \qquad \text{fresh}(a) \qquad \Sigma' = \Sigma\,[a_{l'} \mapsto v]}{\mathcal{E}^{i,l}_{\Sigma}\,[\textbf{new}\ l'\ v] \to \mathcal{E}^{i,l}_{\Sigma'}\,[a_{l'}]}
$$

$$
\text{I-READ}\quad \mathcal{E}^{i,l}_{\Sigma}\,[\textbf{read}\ a_{l'}] \to \mathcal{E}^{i,l \sqcup l'}_{\Sigma}\,[\Sigma(a_{l'})]
$$

$$
\text{I-WRITE}\quad \frac{l \sqsubseteq l' \qquad \Sigma' = \Sigma\,[a_{l'} \mapsto v]}{\mathcal{E}^{i,l}_{\Sigma}\,[\textbf{write}\ a_{l'}\ v] \to \mathcal{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}
$$

$$
\text{I-LABELOF2}\quad \mathcal{E}^{i,l}_{\Sigma}\,[\textbf{labelOf}\ a_{l'}] \to \mathcal{E}^{i,l}_{\Sigma}\,[l']
$$

Fig. 10: Syntax and semantics for labeled references. These rules are understood to be an addition to the existing rules given earlier.

C.3 Clearance

Systems like LIO, COWL, and Breeze additionally provide a discretionary access control (DAC) mechanism—called *clearance*—at the language level [20, 39]. This mechanisms is used to restrict a computation from allocating and accessing data (or communicating with entities) above a specified label, the clearance. Amending our IFC language with clearance is straight forward, and, can be done using our notation of a restricted language. To this end, we first extend tasks to track a clearance label alongside the current label, and amend the core IFC language with two new terminals for retrieving and setting this value. Since this extension only adds a per-task mutable variable whose value has no influence on the system, all security guarantees still hold, by essentially the same proofs. However, this does not implement any DAC mechanism yet. To do so, we can

$$\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet) \text{ (map } \varepsilon_l \ ts)$$

$$\langle \Sigma, e \rangle^i_{l'} \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \varepsilon_l(\Sigma), \varepsilon_l(e) \rangle^i_{l'} & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\textbf{Labeled } l' \ e) = \begin{cases} \textbf{Labeled } l' \bullet & l' \not\sqsubseteq l \\ \textbf{Labeled } l' \ e & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\emptyset) = \emptyset$$

$$\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Sigma\,[a_{l'} \mapsto v]) = \begin{cases} \varepsilon_l(\Sigma)\,[a_{l'} \mapsto \bullet] & l' \not\sqsubseteq l \\ \varepsilon_l(\Sigma)\,[a_{l'} \mapsto \varepsilon_l(v)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Theta) = \Theta \preceq l$$

Fig. 11: Erasure function for the full IFC language, with all extensions. In all cases that are not specified, including target-language constructs, ε_l is applied homomorphically (e.g., $\varepsilon_l(\textbf{setLabel } e) = \textbf{setLabel } \varepsilon_l(e)$). This definition replaces the one from Fig. 5, which is for the IFC language without extensions.

restrict the language with a family of predicates $\mathcal{P}_{\text{clearance}}$: All rules that raise the current label (e.g., I-SETLABEL), perform allocation (e.g., I-SANDBOX and I-send), or set the clearance (clearance should not be arbitrarily raised), a predicate that uses the clearance to impose DAC is used. For instance, the predicate for I-SETLABEL prevents the current label from being raised above the clearance (and thus permit reads above the clearance). The predicate $P := l \sqsubseteq l'$ achieves this restriction, where l' is the clearance and l is the current label. The other predicates are defined in a similar way and omitted for brevity.

C.4 Privileges

Decentralized IFC extends IFC with the decentralized label model of Myers and Liskov [30] to allow for more general applications, including systems consisting of mutually distrustful parties. In a decentralized system, a computation is executed with a set of *privileges*, which, when exercised, allow the computation to declassify data (e.g., by lowering the current label). Practical IFC systems (e.g., [20, 31, 39, 51]) rely on privileges to implement many applications. The challenge with such an extension lies in the precise security guarantees that must be proved, which to the best of our knowledge is an open research problem.

Our implementation for Node.js and COWL both provide privileges, but we have not formalized this part any further.

D Non-Interference Proof

In this section we prove the theorems we have stated in the paper. Note that we prove soundness of the system including the formally defined extensions from Appendix C. We first observe that the non-interference claims for the languages $L_{\text{IFC}}(\text{SEQ}, \lambda)$ and $L_{\text{IFC}}(\text{RR}, \lambda)$ in Theorems 1 and 2 follow directly from Theorem 5, where the set of predicates is the set of always valid predicates (i.e., no restriction).

Before we proceed with the proof of Theorem 5, we state and proof two lemmas we will use.

Lemma 1. *For any task t, task lists ts, store Σ, and label l, if $\varepsilon_l(t) = \bullet$, then there exists a task list ts' and a store Σ' such that*

$$\Sigma; t, ts \hookrightarrow \Sigma'; ts, ts' \tag{3}$$

$$\varepsilon_l(ts') = \textbf{nil} \tag{4}$$

$$\varepsilon_l(\Sigma') = \varepsilon_l(\Sigma) \tag{5}$$

Proof. From $\varepsilon_l(t) = \bullet$ we know that the current label l_{cur} of t must be above l. Furthermore, tasks can always take a step (if no regular rule applies, then I-NOSTEP can be used), and thus we consider all rules that could be applied to execute t.

Case I-noStep and I-done In this case, the task t is dropped, and thus $ts' =$ **nil** and $\Sigma' = \Sigma$ satisfy conditions (4) and (5).

Case I-sandbox The newly created task has a label of at least l_{cur}, and will thus be erased, as required by condition (4). Furthermore, the state only changes for the newly created thread, and thus the state change is erased, showing (5).

In all other rules, no new tasks are created, and thus ts' consists of just the one task t', to which t executed. Since the tasks label can only increase, t' is still erased, showing condition (4). We are left to show condition (5) for the remaining rules.

Case I-send A new message triple with label l' gets added to the message queue of the receiving thread. However, since $l_{\text{cur}} \sqsubseteq l'$, the triple will get erased.

Case I-recv and I-noRecv In this case, only the queue of task t can change, which gets erased.

Case I-new The newly allocated address has to be at a label at least as high as l_{cur}, and will thus be erased.

Case I-write Only addresses with a label l' above l_{cur} can be written, thus the change in Σ_1 will get erased.

Otherwise. None of the other rules modify the state Σ, and thus $\Sigma' = \Sigma$ will trivially satisfy condition (5).

□

Lemma 2. *We consider, for any target language λ, the restricted IFC language $L^{\mathcal{P}}_{IFC}(\alpha, \lambda)$ (according to Definition 6). Then, for any configurations c_1, c_1', c_2, and label l where*

$$c_1 \approx_l c_2 \qquad \textit{and} \qquad c_1 \hookrightarrow c_1' \tag{6}$$

there exists a configuration c_2' such that

$$c_1' \approx_l c_2' \qquad \textit{and} \qquad c_2 \hookrightarrow^* c_2' \ . \tag{7}$$

Proof. First, we observe there must be at least one task in c_1, otherwise it could not take a step. Thus, c_1 is of the form $\Sigma_1; t_1, ts_1$. Furthermore, let c_2 be $\Sigma_2; ts_2$. Consider two cases:

- $\varepsilon_l(t_1) = \bullet$. By the definition of ε_l, we know that $l \sqsubseteq l_{\text{cur}}$ where l_{cur} is the label of t_1. In this case, we do not need to take a step for c_2, because $c_2' = c_2$ will already be l-equivalent to c_1'. To show this, note that the tasks ts_1 in c_1 are left in the same order and unmodified (the scheduling policy only modifies the first task). The task t_1 either gets dropped (by I-NoStep), or transforms into a task t_1' as well as potentially spawning a new task t_1''. Since both t_1' and t_1'' have a label that is at least as high as the label of t_1 (can be seen by inspecting all reduction rules), they will get filtered by ε_l in c_1'. Therefore, the l-equivalence of the task list is guaranteed. Lets consider the possible changes to Σ_1: Only five reduction interact with Σ_1, thus it suffices to consider these cases:
  Case I-send A new message triple with label l' gets added to the message queue of the receiving thread. However, since $l_{\text{cur}} \sqsubseteq l'$, the triple will get erased.
  Case I-recv and I-noRecv In this case, only the queue of task t_1 can change, which gets erased.
  Case I-new The newly allocated address has to be at a label at least as high as l_{cur}, and will thus be erased.
  Case I-write Only addresses with a label l' above l_{cur} can be written, thus the change in Σ_1 will get erased.
  This ensures that $c_1' \approx_l c_2'$, as well as $c_2 \hookrightarrow^* c_2'$ (in zero steps), as claimed.
- $\varepsilon_l(t_1) \neq \bullet$. By the definition of ε_l, the task list ts_2 in c_2 must be of the form ts_2', t_2, ts_2'' (for some task lists ts_2', ts_2'' and some task t_2) where

$$\varepsilon_l(ts_2') = \mathbf{nil} \tag{8}$$
$$\varepsilon_l(t_2) = \varepsilon_l(t_1) \tag{9}$$
$$\varepsilon_l(ts_2'') = \varepsilon_l(ts_1) \tag{10}$$

  (where **nil** is the empty list of tasks). Now, intuitively we will first execute a number of steps to process the tasks in ts_2' (execute them one step and move them to the back of the task list, or drop them if they are done or stuck). Then, the task t_2 can take the same step as t_1, which will result in a configuration c_2' with the desired properties. More formally, we can proceed as follows:

First, we can apply Lemma 1 continuously for all the task in ts_2', until we reach a configuration $c_2'' = \Sigma_2'; t_2, ts_2'', ts_2'''$ for some ts_2''' such that $\varepsilon_l(ts_2''') = \textbf{nil}$ and $\varepsilon_l(\Sigma_2) = \varepsilon_l(\Sigma_2')$. We note that $\varepsilon_l(c_1) = \varepsilon_l(c_2'')$ (by the definition of ε_l).

Now, the first task t_2 in c_2'' is l-equivalent to the task t_1. This implies that the two tasks must have the same id, label and can only differ in the expression or store if some subexpression is of the form **Labeled** l' e. In this case, the expression e could be different in the two threads if $l_{\text{cur}} \sqsubseteq l'$. However, none of the reduction rules depend on an expression in that position, and there is never a hole in that position where evaluation could take place. Thus, the same rules will syntactically match for both task, and we are left to argue that all premises evaluate to the same values for t_1 and t_2, as well as that the resulting states Σ_1' and Σ_2'' are l-equivalent. The additional premises P that follow the condition in Definition 6 are not a problem, since those predicates only depend on $\varepsilon_l(c_1)$, which is equivalent to $\varepsilon_l(c_2'')$, and thus those predicates evaluate in the same way. All other premises are either on the threads labels (which are the same), or on the state Σ_1, or Σ_2', respectively. Because $\varepsilon_l(\Sigma_1) = \varepsilon_l(\Sigma_2')$, all of these also evaluate in the same way, as can be seen by simply considering all rules that involve or change the state:

Case I-send Here, the task t_2 will send the same message to the same receiver queue. This queue is either completely erased, or it is l-equivalent. In both cases, l-equivalence of Σ_1' and Σ_2' is preserved.

Case I-recv and I-noRecv When the tasks are receiving a message, then by the reduction rules we know that they first filter the queue by the label l_{cur} of t_1. We also know that the queues are equivalent when filtered by the less restrictive label l, thus the messages received (or dropped) from the queue are equivalent.

Case I-new The newly allocated address can be the same for both t_1 and t_2, thus resulting in l-equivalent states.

Case I-write By $\varepsilon_l(t_1) = \mathit{earse}\ l\ t_2$ both tasks write the same value, and therefore the resulting states will still be l-equivalent.

After t_2 has taken a step, we finally arrive in the desired configuration $c_2' = \Sigma_2''; ts_2'', ts_2''', ts_2''''$, where ts_2'''' contains the task resulting from executing t_2 (and might contain, zero (if the task was done or stuck), one (for most steps) or two tasks if a new task was launched). As required, we have

$$c_2 \hookrightarrow^* c_2'' \hookrightarrow c_2' \quad \wedge \quad c_1' \approx_l c_2'.$$

□

With this, it is easy to proof Theorem 5 as follows.

Proof (Proof of Theorem 5, TSNI). We proof the theorem by induction on the length of the derivation sequence in (1). The base case for derivations of length 0 is trivial, allowing us to simple chose $c_2' = c_2$. In the step case, we assume the theorem holds for derivation sequences of length up to n, and show that it

also holds for those of length $n+1$. We split the derivation sequence from (1) as follows:

$$c_1 \hookrightarrow c_1'' \hookrightarrow^n c_1'$$

for some configuration c_1''. By Lemma 2, we get c'' with

$$c_1'' \approx_l c_2'' \qquad \text{and} \qquad c_2 \hookrightarrow^* c_2'' \tag{11}$$

Applying the induction hypothesis to $c_1'' \hookrightarrow^n c_1'$, we get c_2' with

$$c_1' \approx_l c_2' \qquad \text{and} \qquad c_2'' \hookrightarrow^* c_2' \tag{12}$$

Stitching together the derivation sequences from (11) and (12) directly gives us the right-hand side of the implication in the TSNI definition (2), which concludes the proof. □